Suchen (in Arrays)

Problem

Gegeben: eine endliche Menge von Elementen
(z.B.: Zahlen, Strings, ...)

Frage: kommt ein bestimmtes Element in dieser Menge vor?

Beispiele:

graphischer Dateimanager:
Menge der momentan markierten Dateien

Durchlaufen eines Labyrinths:
Menge der schon betretenen Felder
(um zu verhindern, daß man im Kreis läuft)

Problem

Variante des Problems:

Element = Schlüsselfeld + weitere Komponenten.

Frage: kommt ein Element in der Menge vor, bei dem das Schlüsselfeld einen bestimmten Wert besitzt?
(Wenn ja, liefere das ganze Element zurück.)

Beispiele:

Adreßliste:
(Schlüssel: Name, gesucht: komplette Adresse)

Bezeichnerliste in einem Compiler:
(Schlüssel: Bezeichner, gesucht: Deklaration des Bezeichners)

Ein möglicher Algorithmus

Idee: Array mit Füllstandanzeige

zwei Möglichkeiten:

Füllstand m = Anzahl der Einträge
dann: Einträge von 0 bis $m - 1$.

Füllstand m = Index des letzten Elements
dann: Anzahl der Einträge = $m + 1$.

hier: zweite Möglichkeit.

Ein möglicher Algorithmus: Lineare Suche

Iterative Lösung:

```
public static boolean
enthaelt(int[] array, int fuellstand, int elem) {
    int i;

    for (i = 0; i <= fuellstand; i++) {
        if (array[i] == elem) {
            return true;
        }
    }
    return false;
}
```


Ein möglicher Algorithmus: Lineare Suche

Noch eine iterative Lösung:

```
public static boolean
enthaelt(int[] array, int fuellstand, int elem) {
    int i = 0;

    while (i <= fuellstand && array[i] != elem) {
        i++;
    }
    return (i <= fuellstand);
}
```


Ein möglicher Algorithmus: Lineare Suche

Rekursive Lösung (Teil 1):

```
public static boolean
enthaelt(int[] array, int fuellstand, int elem) {
    return enthZwischen(array, 0, fuellstand, elem);
}
```


Ein möglicher Algorithmus: Lineare Suche

Rekursive Lösung (Teil 2):

```
public static boolean
enthZwischen(int[] array, int unten, int oben, int elem) {
// existiert ein Index i, so daß
// unten <= i <= oben  und  array[i] == elem?
    if (unten > oben) {
        return false;
    } else if (array[unten] == elem) {
        return true;
    } else {
        return enthZwischen(array, unten + 1, oben, elem);
    }
}
```

Zeitverbrauch

n = Anzahl der Elemente der Menge = `Fuellstand + 1`.

t_1 = Zeit für Initialisierung (vor Schleifenbeginn).

t_2 = (Maximal-)Zeit für Überprüfung der Schleifenbedingung.

t_3 = Zeit für Schleifenrumpf.

t_4 = Zeit für Beendigung der Funktion (nach Schleifenende).

Dann ist die Laufzeit höchstens $n \cdot (t_2 + t_3) + (t_1 + t_2 + t_4)$.

Mathematische Überlegungen

Laufzeitabschätzungen enthalten in der Regel unbekannte Konstanten.

Kann man trotzdem

- daraus etwas über das Verhalten eines Algorithmus ableiten?
- Algorithmen vergleichen?

Mathematische Überlegungen

Meist am interessantesten:

Wie verhält sich ein Algorithmus bei *großen* Eingaben?

Mathematisch ausgedrückt:

Wie verhält sich die Laufzeitfunktion für $n \to \infty$?

Mathematische Überlegungen

Fall 1: Polynome

$p(n) = c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + \ldots + c_0$

k = höchster Exponent = Grad des Polynoms.

c_k = führender Koeffizient (hier immer positiv).

Das Verhalten eines Polynoms für $n \to \infty$ hängt ab
in erster Linie von seinem Grad,
in zweiter Linie vom führenden Koeffizienten,
und erst danach vom Rest des Polynoms.

Mathematische Überlegungen

Falls Grad von $p_1 >$ Grad von p_2,
dann existiert eine Zahl m,
so daß $p_1(n) > p_2(n)$ für alle $n > m$.

Beispiel:

Sei $p_1(n) = n^2$,
$p_2(n) = 100\,n + 20000$.

Dann ist für $n > 200$:

$p_1(n) = n^2 > 200\,n = 100\,n + 100\,n > 100\,n + 20000 = p_2(n)$.

Mathematische Überlegungen

Fall 2: Exponentialfunktionen

Eine Exponentialfunktion c^n (mit $c > 1$ konstant)
wächst (viel!) schneller als *jedes* Polynom:

Für jedes Polynom $p(n)$ existiert eine Zahl m,
so daß $c^n > p(n)$ für alle $n > m$.

Mathematische Überlegungen

Zum Vergleich:

n	n^2	n^3	n^4	2^n
10	100	1000	10000	1024
20	400	8000	160000	1048576
30	900	27000	810000	1073741824
40	1600	64000	2560000	1099511627776
50	2500	125000	6250000	1125899906842624
60	3600	216000	12960000	1152921504606846976
70	4900	343000	24010000	1180591620717411303424
80	6400	512000	40960000	1208925819614629174706176

Mathematische Überlegungen

Fall 3: Logarithmen

$c^m = n \quad \Rightarrow \quad \log_c n = m.$

Eine Logarithmusfunktion $c_0 \log_c n$ (mit $c > 1$ und $c_0 > 1$ konstant)
wächst (viel!) langsamer als jedes nicht-konstante Polynom.

Die Basis des Logarithmus spielt fast keine Rolle:
Logarithmusfunktionen $\log_c n$ und $\log_d n$ unterscheiden sich nur durch einen konstanten Faktor.

Mathematische Überlegungen

Zum Vergleich:

n	$\log_2 n$
1024	10
1048576	20
1073741824	30
1099511627776	40
1125899906842624	50
1152921504606846976	60
1180591620717411303424	70
1208925819614629174706176	80

Mathematische Überlegungen

O-Notation:

Sei $f(n)$ eine Funktion.

$O(f(n))$ = Menge aller Funktionen, die höchstens so schnell wachsen wie $c \cdot f(n)$ (für irgendeine Konstante c).

Formal: $g(n) \in O(f(n))$, falls natürliche Zahlen m und c existieren, so daß $g(n) \leq c \cdot f(n)$ für alle $n > m$.

Beispiel:

$g(n) = 3n^2 + 4n + 5$.

$g(n) \in O(n^2)$, denn für $n > 1$ gilt:
$g(n) = 3n^2 + 4n + 5 < 3n^2 + 4n^2 + 5n^2 = 12n^2$.

Mathematische Überlegungen

Hierarchie:

$$\begin{aligned} O(1) &\subset \ldots \subset O(\log(\log n)) \subset O(\log n) \\ &\subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots \\ &\subset O(2^n) \subset O(3^n) \subset \ldots \\ &\subset O(2^{2^n}) \subset O(2^{2^{2^n}}) \subset \ldots \end{aligned}$$

Suchen: Ein anderer Algorithmus

Lineare Suche in einem Array mit n Elementen braucht $O(n)$ Zeit.

Geht es auch schneller?

Ja, falls das Array *sortiert* ist.

Ein anderer Algorithmus: Binäre Suche

Iterative Lösung:

```
public static boolean
enthaelt(int[] array, int fuellstand, int elem) {
    int unten = 0;
    int oben = fuellstand;
    int mitte;

    ...
```


Ein anderer Algorithmus: Binäre Suche

```
    while (unten <= oben) {
        mitte = (oben + unten) / 2;
        if (array[mitte] == elem) {
            return true;
        } else if (array[mitte] > elem) {
            oben = mitte - 1;
        } else {
            unten = mitte + 1;
        }
    }
    return false;
}
```


Ein anderer Algorithmus: Binäre Suche

Rekursive Lösung:

```
public static boolean
enthaelt(int[] array, int fuellstand, int elem) {
    return enthZwischen(array, 0, fuellstand, elem);
}


public static boolean
enthZwischen(int[] array, int unten, int oben, int elem) {
// existiert ein Index i, so daß
// unten <= i <= oben  und  array[i] == elem?

    ...
```


Ein anderer Algorithmus: Binäre Suche

```
public static boolean
enthZwischen(int[] array, int unten, int oben, int elem) {
    int mitte = (oben + unten) / 2;
    if (unten > oben) {
        return false;
    } else if (array[mitte] == elem) {
        return true;
    } else if (array[mitte] > elem) {
        return enthZwischen(array, unten, mitte - 1, elem);
    } else {
        return enthZwischen(array, mitte + 1, oben, elem);
    }
}
```

Ein anderer Algorithmus: Binäre Suche

Abschätzung des Zeitverbrauchs:

Anz. der Elem. zwischen `oben` und `unten` (`oben` $-$ `unten` $+ 1$)	Anz. der Iterationen (höchstens)
0	0
1	1
3	2
7	3
15	4
$2^m - 1$	m
n	$\lceil \log_2(n+1) \rceil$

Vergleich

Lineare Suche in unsortiertem Array: $O(n)$.

Binäre Suche in sortiertem Array: $O(\log n)$.

Aber ...

Pferdefuß: Einfügen eines neuen Elements

bei unsortierten Arrays:

Füllstand inkrementieren;
neues Element oben eintragen.
$\rightsquigarrow O(1)$.

bei sortierten Arrays:

Füllstand inkrementieren;
neues Element *an der richtigen Stelle* einfügen;
vorher: alle Elemente darüber um eine Position verschieben.
$\rightsquigarrow O(n)$.

Aber ...

Meist tritt beides kombiniert auf:
überprüfen, ob vorhanden, und falls nicht, einfügen.

bei unsortierten Arrays: $O(n) + O(1) = O(n)$.

bei sortierten Arrays: $O(\log n) + O(n) = O(n)$.

(bessere Verfahren: demnächst)